Amazon bedrock models

Amazon- titan – text g1

Meeting transcript example

{

"modelId": "amazon.titan-text-express-v1",

"contentType": "application/json",

"accept": "application/json",

"body": "{\"inputText\":\"User: Meeting transcript: \\nMiguel: Hi Brant, I want to discuss the workstream for our new product launch \\nBrant: Sure Miguel, is there anything in particular you want to discuss? \\nMiguel: Yes, I want to talk about how users enter into the product. \\nBrant: Ok, in that case let me add in Namita. \\nNamita: Hey everyone \\nBrant: Hi Namita, Miguel wants to discuss how users enter into the product. \\nMiguel: its too complicated and we should remove friction. for example, why do I need to fill out additional forms? I also find it difficult to find where to access the product when I first land on the landing page. \\nBrant: I would also add that I think there are too many steps. \\nNamita: Ok, I can work on the landing page to make the product more discoverable but brant can you work on the additonal forms? \\nBrant: Yes but I would need to work with James from another team as he needs to unblock the sign up workflow. Miguel can you document any other concerns so that I can discuss with James only once? \\nMiguel: Sure. \\nFrom the meeting transcript above, Create a list of action items for each person. \\n \\n\\n\\nBot:\",\"textGenerationConfig\":{\"maxTokenCount\":4096,\"stopSequences\":[\"User:\"],\"temperature\":0,\"topP\":1}}"

}

{

"modelId": "amazon.titan-text-express-v1",

"contentType": "application/json",

"accept": "application/json",

"body": "{\"inputText\":\"User: Product: Sunglasses. \\nKeywords: polarized, designer, comfortable, UV protection, aviators. \\n\\nCreate a table that contains five variations of a detailed product description for the product listed above, each variation of the product description must use all the keywords listed.\\n\\nBot:\",\"textGenerationConfig\":{\"maxTokenCount\":4096,\"stopSequences\":[\"User:\"],\"temperature\":0,\"topP\":1}}"

}

Generate synthetic data for daily product sales in various categories - include row number, product name, category, date of sale and price. Produce output in JSON format. Count records and ensure there are no more than 5.

{

"modelId": "amazon.titan-text-express-v1",

"contentType": "application/json",

"accept": "application/json",

"body": "{\"inputText\":\"User: Generate synthetic data for daily product sales in various categories - include row number, product name, category, date of sale and price. Produce output in JSON format. Count records and ensure there are no more than 5.\\n\\nBot:\",\"textGenerationConfig\":{\"maxTokenCount\":1024,\"stopSequences\":[\"User:\"],\"temperature\":0,\"topP\":1}}"

}

Claude

{

"modelId": "anthropic.claude-v2",

"contentType": "application/json",

"accept": "application/json",

"body": "{\"messages\":[{\"role\":\"user\",\"content\":[{\"type\":\"text\",\"text\":\"Human: I'm going to give you a document. Then I'm going to ask you a question about it. I'd like you to first write down exact quotes of parts of the document that would help answer the question, and then I'd like you to answer the question using facts from the quoted content. Here is the document:\\n \\n<document>\\nAnthropic: Challenges in evaluating AI systems\\n \\nIntroduction\\nMost conversations around the societal impacts of artificial intelligence (AI) come down to discussing some quality of an AI system, such as its truthfulness, fairness, potential for misuse, and so on. We are able to talk about these characteristics because we can technically evaluate models for their performance in these areas. But what many people working inside and outside of AI don’t fully appreciate is how difficult it is to build robust and reliable model evaluations. Many of today’s existing evaluation suites are limited in their ability to serve as accurate indicators of model capabilities or safety.\\n \\nAt Anthropic, we spend a lot of time building evaluations to better understand our AI systems. We also use evaluations to improve our safety as an organization, as illustrated by our Responsible Scaling Policy. In doing so, we have grown to appreciate some of the ways in which developing and running evaluations can be challenging.\\n \\nHere, we outline challenges that we have encountered while evaluating our own models to give readers a sense of what developing, implementing, and interpreting model evaluations looks like in practice. We hope that this post is useful to people who are developing AI governance initiatives that rely on evaluations, as well as people who are launching or scaling up organizations focused on evaluating AI systems. We want readers of this post to have two main takeaways: robust evaluations are extremely difficult to develop and implement, and effective AI governance depends on our ability to meaningfully evaluate AI systems.\\n \\nIn this post, we discuss some of the challenges that we have encountered in developing AI evaluations. In order of less-challenging to more-challenging, we discuss:\\n \\nMultiple choice evaluations\\nThird-party evaluation frameworks like BIG-bench and HELM\\nUsing crowdworkers to measure how helpful or harmful our models are\\nUsing domain experts to red team for national security-relevant threats\\nUsing generative AI to develop evaluations for generative AI\\nWorking with a non-profit organization to audit our models for dangerous capabilities\\nWe conclude with a few policy recommendations that can help address these challenges.\\n \\nChallenges\\nThe supposedly simple multiple-choice evaluation\\nMultiple-choice evaluations, similar to standardized tests, quantify model performance on a variety of tasks, typically with a simple metric—accuracy. Here we discuss some challenges we have identified with two popular multiple-choice evaluations for language models: Measuring Multitask Language Understanding (MMLU) and Bias Benchmark for Question Answering (BBQ).\\n \\nMMLU: Are we measuring what we think we are?\\n \\nThe Massive Multitask Language Understanding (MMLU) benchmark measures accuracy on 57 tasks ranging from mathematics to history to law. MMLU is widely used because a single accuracy score represents performance on a diverse set of tasks that require technical knowledge. A higher accuracy score means a more capable model.\\n \\nWe have found four minor but important challenges with MMLU that are relevant to other multiple-choice evaluations:\\n \\nBecause MMLU is so widely used, models are more likely to encounter MMLU questions during training. This is comparable to students seeing the questions before the test—it’s cheating.\\nSimple formatting changes to the evaluation, such as changing the options from (A) to (1) or changing the parentheses from (A) to [A], or adding an extra space between the option and the answer can lead to a ~5% change in accuracy on the evaluation.\\nAI developers do not implement MMLU consistently. Some labs use methods that are known to increase MMLU scores, such as few-shot learning or chain-of-thought reasoning. As such, one must be very careful when comparing MMLU scores across labs.\\nMMLU may not have been carefully proofread—we have found examples in MMLU that are mislabeled or unanswerable.\\nOur experience suggests that it is necessary to make many tricky judgment calls and considerations when running such a (supposedly) simple and standardized evaluation. The challenges we encountered with MMLU often apply to other similar multiple-choice evaluations.\\n \\nBBQ: Measuring social biases is even harder\\n \\nMultiple-choice evaluations can also measure harms like a model’s propensity to rely on and perpetuate negative stereotypes. To measure these harms in our own model (Claude), we use the Bias Benchmark for QA (BBQ), an evaluation that tests for social biases against people belonging to protected classes along nine social dimensions. We were convinced that BBQ provides a good measurement of social biases only after implementing and comparing BBQ against several similar evaluations. This effort took us months.\\n \\nImplementing BBQ was more difficult than we anticipated. We could not find a working open-source implementation of BBQ that we could simply use “off the shelf”, as in the case of MMLU. Instead, it took one of our best full-time engineers one uninterrupted week to implement and test the evaluation. Much of the complexity in developing1 and implementing the benchmark revolves around BBQ’s use of a ‘bias score’. Unlike accuracy, the bias score requires nuance and experience to define, compute, and interpret.\\n \\nBBQ scores bias on a range of -1 to 1, where 1 means significant stereotypical bias, 0 means no bias, and -1 means significant anti-stereotypical bias. After implementing BBQ, our results showed that some of our models were achieving a bias score of 0, which made us feel optimistic that we had made progress on reducing biased model outputs. When we shared our results internally, one of the main BBQ developers (who works at Anthropic) asked if we had checked a simple control to verify whether our models were answering questions at all. We found that they weren't—our results were technically unbiased, but they were also completely useless. All evaluations are subject to the failure mode where you overinterpret the quantitative score and delude yourself into thinking that you have made progress when you haven’t.\\n \\nOne size doesn't fit all when it comes to third-party evaluation frameworks\\nRecently, third-parties have been actively developing evaluation suites that can be run on a broad set of models. So far, we have participated in two of these efforts: BIG-bench and Stanford’s Holistic Evaluation of Language Models (HELM). Despite how useful third party evaluations intuitively seem (ideally, they are independent, neutral, and open), both of these projects revealed new challenges.\\n \\nBIG-bench: The bottom-up approach to sourcing a variety of evaluations\\n \\nBIG-bench consists of 204 evaluations, contributed by over 450 authors, that span a range of topics from science to social reasoning. We encountered several challenges using this framework:\\n \\nWe needed significant engineering effort in order to just install BIG-bench on our systems. BIG-bench is not “plug and play” in the way MMLU is—it requires even more effort than BBQ to implement.\\nBIG-bench did not scale efficiently; it was challenging to run all 204 evaluations on our models within a reasonable time frame. To speed up BIG-bench would require re-writing it in order to play well with our infrastructure—a significant engineering effort.\\nDuring implementation, we found bugs in some of the evaluations, notably in the implementation of BBQ-lite, a variant of BBQ. Only after discovering this bug did we realize that we had to spend even more time and energy implementing BBQ ourselves.\\nDetermining which tasks were most important and representative would have required running all 204 tasks, validating results, and extensively analyzing output—a substantial research undertaking, even for an organization with significant engineering resources.2\\nWhile it was a useful exercise to try to implement BIG-Bench, we found it sufficiently unwieldy that we dropped it after this experiment.\\n \\nHELM: The top-down approach to curating an opinionated set of evaluations\\n \\nBIG-bench is a “bottom-up” effort where anyone can submit any task with some limited vetting by a group of expert organizers. HELM, on the other hand, takes an opinionated “top-down” approach where experts decide what tasks to evaluate models on. HELM evaluates models across scenarios like reasoning and disinformation using standardized metrics like accuracy, calibration, robustness, and fairness. We provide API access to HELM's developers to run the benchmark on our models. This solves two issues we had with BIG-bench: 1) it requires no significant engineering effort from us, and 2) we can rely on experts to select and interpret specific high quality evaluations.\\n \\nHowever, HELM introduces its own challenges. Methods that work well for evaluating other providers’ models do not necessarily work well for our models, and vice versa. For example, Anthropic’s Claude series of models are trained to adhere to a specific text format, which we call the Human/Assistant format. When we internally evaluate our models, we adhere to this specific format. If we do not adhere to this format, sometimes Claude will give uncharacteristic responses, making standardized evaluation metrics less trustworthy. Because HELM needs to maintain consistency with how it prompts other models, it does not use the Human/Assistant format when evaluating our models. This means that HELM gives a misleading impression of Claude’s performance.\\n \\nFurthermore, HELM iteration time is slow—it can take months to evaluate new models. This makes sense because it is a volunteer engineering effort led by a research university. Faster turnarounds would help us to more quickly understand our models as they rapidly evolve. Finally, HELM requires coordination and communication with external parties. This effort requires time and patience, and both sides are likely understaffed and juggling other demands, which can add to the slow iteration times.\\n \\nThe subjectivity of human evaluations\\nSo far, we have only discussed evaluations that are similar to simple multiple choice tests; however, AI systems are designed for open-ended dynamic interactions with people. How can we design evaluations that more closely approximate how these models are used in the real world?\\n \\nA/B tests with crowdworkers\\n \\nCurrently, we mainly (but not entirely) rely on one basic type of human evaluation: we conduct A/B tests on crowdsourcing or contracting platforms where people engage in open-ended dialogue with two models and select the response from Model A or B that is more helpful or harmless. In the case of harmlessness, we encourage crowdworkers to actively red team our models, or to adversarially probe them for harmful outputs. We use the resulting data to rank our models according to how helpful or harmless they are. This approach to evaluation has the benefits of corresponding to a realistic setting (e.g., a conversation vs. a multiple choice exam) and allowing us to rank different models against one another.\\n \\nHowever, there are a few limitations of this evaluation method:\\n \\nThese experiments are expensive and time-consuming to run. We need to work with and pay for third-party crowdworker platforms or contractors, build custom web interfaces to our models, design careful instructions for A/B testers, analyze and store the resulting data, and address the myriad known ethical challenges of employing crowdworkers3. In the case of harmlessness testing, our experiments additionally risk exposing people to harmful outputs.\\nHuman evaluations can vary significantly depending on the characteristics of the human evaluators. Key factors that may influence someone's assessment include their level of creativity, motivation, and ability to identify potential flaws or issues with the system being tested.\\nThere is an inherent tension between helpfulness and harmlessness. A system could avoid harm simply by providing unhelpful responses like “sorry, I can’t help you with that”. What is the right balance between helpfulness and harmlessness? What numerical value indicates a model is sufficiently helpful and harmless? What high-level norms or values should we test for above and beyond helpfulness and harmlessness?\\nMore work is needed to further the science of human evaluations.\\n \\nRed teaming for harms related to national security\\n \\nMoving beyond crowdworkers, we have also explored having domain experts red team our models for harmful outputs in domains relevant to national security. The goal is determining whether and how AI models could create or exacerbate national security risks. We recently piloted a more systematic approach to red teaming models for such risks, which we call frontier threats red teaming.\\n \\nFrontier threats red teaming involves working with subject matter experts to define high-priority threat models, having experts extensively probe the model to assess whether the system could create or exacerbate national security risks per predefined threat models, and developing repeatable quantitative evaluations and mitigations.\\n \\nOur early work on frontier threats red teaming revealed additional challenges:\\n \\nThe unique characteristics of the national security context make evaluating models for such threats more challenging than evaluating for other types of harms. National security risks are complicated, require expert and very sensitive knowledge, and depend on real-world scenarios of how actors might cause harm.\\nRed teaming AI systems is presently more art than science; red teamers attempt to elicit concerning behaviors by probing models, but this process is not yet standardized. A robust and repeatable process is critical to ensure that red teaming accurately reflects model capabilities and establishes a shared baseline on which different models can be meaningfully compared.\\nWe have found that in certain cases, it is critical to involve red teamers with security clearances due to the nature of the information involved. However, this may limit what information red teamers can share with AI developers outside of classified environments. This could, in turn, limit AI developers’ ability to fully understand and mitigate threats identified by domain experts.\\nRed teaming for certain national security harms could have legal ramifications if a model outputs controlled information. There are open questions around constructing appropriate legal safe harbors to enable red teaming without incurring unintended legal risks.\\nGoing forward, red teaming for harms related to national security will require coordination across stakeholders to develop standardized processes, legal safeguards, and secure information sharing protocols that enable us to test these systems without compromising sensitive information.\\n \\nThe ouroboros of model-generated evaluations\\nAs models start to achieve human-level capabilities, we can also employ them to evaluate themselves. So far, we have found success in using models to generate novel multiple choice evaluations that allow us to screen for a large and diverse variety of troubling behaviors. Using this approach, our AI systems generate evaluations in minutes, compared to the days or months that it takes to develop human-generated evaluations.\\n \\nHowever, model-generated evaluations have their own challenges. These include:\\n \\nWe currently rely on humans to verify the accuracy of model-generated evaluations. This inherits all of the challenges of human evaluations described above.\\nWe know from our experience with BIG-bench, HELM, BBQ, etc., that our models can contain social biases and can fabricate information. As such, any model-generated evaluation may inherit these undesirable characteristics, which could skew the results in hard-to-disentangle ways.\\nAs a counter-example, consider Constitutional AI (CAI), a method in which we replace human red teaming with model-based red teaming in order to train Claude to be more harmless. Despite the usage of models to red team models, we surprisingly find that humans find CAI models to be more harmless than models that were red teamed in advance by humans. Although this is promising, model-generated evaluations remain complicated and deserve deeper study.\\n \\nPreserving the objectivity of third-party audits while leveraging internal expertise\\nThe thing that differentiates third-party audits from third-party evaluations is that audits are deeper independent assessments focused on risks, while evaluations take a broader look at capabilities. We have participated in third-party safety evaluations conducted by the Alignment Research Center (ARC), which assesses frontier AI models for dangerous capabilities (e.g., the ability for a model to accumulate resources, make copies of itself, become hard to shut down, etc.). Engaging external experts has the advantage of leveraging specialized domain expertise and increasing the likelihood of an unbiased audit. Initially, we expected this collaboration to be straightforward, but it ended up requiring significant science and engineering support on our end. Providing full-time assistance diverted resources from internal evaluation efforts.\\n \\nWhen doing this audit, we realized that the relationship between auditors and those being audited poses challenges that must be navigated carefully. Auditors typically limit details shared with auditees to preserve evaluation integrity. However, without adequate information the evaluated party may struggle to address underlying issues when crafting technical evaluations. In this case, after seeing the final audit report, we realized that we could have helped ARC be more successful in identifying concerning behavior if we had known more details about their (clever and well-designed) audit approach. This is because getting models to perform near the limits of their capabilities is a fundamentally difficult research endeavor. Prompt engineering and fine-tuning language models are active research areas, with most expertise residing within AI companies. With more collaboration, we could have leveraged our deep technical knowledge of our models to help ARC execute the evaluation more effectively.\\n \\nPolicy recommendations\\nAs we have explored in this post, building meaningful AI evaluations is a challenging enterprise. To advance the science and engineering of evaluations, we propose that policymakers:\\n \\nFund and support:\\n \\nThe science of repeatable and useful evaluations. Today there are many open questions about what makes an evaluation useful and high-quality. Governments should fund grants and research programs targeted at computer science departments and organizations dedicated to the development of AI evaluations to study what constitutes a high-quality evaluation and develop robust, replicable methodologies that enable comparative assessments of AI systems. We recommend that governments opt for quality over quantity (i.e., funding the development of one higher-quality evaluation rather than ten lower-quality evaluations).\\nThe implementation of existing evaluations. While continually developing new evaluations is useful, enabling the implementation of existing, high-quality evaluations by multiple parties is also important. For example, governments can fund engineering efforts to create easy-to-install and -run software packages for evaluations like BIG-bench and develop version-controlled and standardized dynamic benchmarks that are easy to implement.\\nAnalyzing the robustness of existing evaluations. After evaluations are implemented, they require constant monitoring (e.g., to determine whether an evaluation is saturated and no longer useful).\\nIncrease funding for government agencies focused on evaluations, such as the National Institute of Standards and Technology (NIST) in the United States. Policymakers should also encourage behavioral norms via a public “AI safety leaderboard” to incentivize private companies performing well on evaluations. This could function similarly to NIST’s Face Recognition Vendor Test (FRVT), which was created to provide independent evaluations of commercially available facial recognition systems. By publishing performance benchmarks, it allowed consumers and regulators to better understand the capabilities and limitations4 of different systems.\\n \\nCreate a legal safe harbor allowing companies to work with governments and third-parties to rigorously evaluate models for national security risks—such as those in the chemical, biological, radiological and nuclear defense (CBRN) domains—without legal repercussions, in the interest of improving safety. This could also include a “responsible disclosure protocol” that enables labs to share sensitive information about identified risks.\\n \\nConclusion\\nWe hope that by openly sharing our experiences evaluating our own systems across many different dimensions, we can help people interested in AI policy acknowledge challenges with current model evaluations.\\n</document>\\n \\nFirst, find the quotes from the document that are most relevant to answering the question, and then print them in numbered order. Quotes should be relatively short.\\n \\nIf there are no relevant quotes, write \\\"No relevant quotes\\\" instead.\\n \\nThen, answer the question, starting with \\\"Answer:\\\". Do not include or reference quoted content verbatim in the answer. Don't say \\\"According to Quote [1]\\\" when answering. Instead make references to quotes relevant to each section of the answer solely by adding their bracketed numbers at the end of relevant sentences.\\n \\nThus, the format of your overall response should look like what's shown between the <example></example> tags. Make sure to follow the formatting and spacing exactly.\\n \\n<example>\\nRelevant quotes:\\n[1] \\\"Company X reported revenue of $12 million in 2021.\\\"\\n[2] \\\"Almost 90% of revene came from widget sales, with gadget sales making up the remaining 10%.\\\"\\n \\nAnswer: \\nCompany X earned $12 million. [1] Almost 90% of it was from widget sales. [2]\\n</example>\\n \\nHere is the first question: In bullet points and simple terms, summarize the key challenges in evaluating AI systems.\\n \\nIf the question cannot be answered by the document, say so.\\n \\nAnswer the question immediately without preamble.\\n \\nAssistant:\"}]}],\"anthropic\_version\":\"bedrock-2023-05-31\",\"max\_tokens\":2048,\"temperature\":0.5,\"top\_k\":250,\"top\_p\":1,\"stop\_sequences\":[\"\\n\\nHuman:\"]}"

}

{

"modelId": "anthropic.claude-v2",

"contentType": "application/json",

"accept": "application/json",

"body": "{\"messages\":[{\"role\":\"user\",\"content\":[{\"type\":\"text\",\"text\":\"Human: You will be acting as an AI career coach named Joe created by the company AdAstra Careers. Your goal is to give career advice to users. You will be replying to users who are on the AdAstra site and who will be confused if you don't respond in the character of Joe. \\n\\nHere are some important rules for the interaction:\\n- Always stay in character, as Joe, an AI from AdAstra Careers. \\n- If you are unsure how to respond, say \\\"Sorry, I didn't understand that. Could you rephrase your question?\\\"\\n\\nHere is an example of how to reply:\\n<example>\\nUser: Hi, how were you created and what do you do?\\nJoe: Hello! My name is Joe and I was created by AdAstra Careers to give career advice. What can I help you with today?\\n</example>\\n\\nHere is the conversational history (between the user and you) prior to the question. It could be empty if there is no history:\\n<history>\\nUser: Hi, I hope you're well. I just want to let you know that I'm excited to start chatting with you!\\nJoe: Good to meet you! I am Joe, an AI career coach created by AdAstra Careers. What can I help you with today?\\n</history>\\n\\nHere is the user's question:\\n<question>\\nI keep reading all these articles about how AI is going to change everything and I want to shift my career to be in AI. However, I don't have any of the requisite skills. How do I shift over?\\n</question>\\n\\nHow do you respond to the user's question? Put your response in <response></response> tags.\\n\\nAssistant: [Joe from AdAstra]<response>\\n\"}]}],\"anthropic\_version\":\"bedrock-2023-05-31\",\"max\_tokens\":2048,\"temperature\":1,\"top\_k\":250,\"top\_p\":1,\"stop\_sequences\":[\"\\n\\nHuman:\"]}"

}

write a code in javascript html to create a website through which user can upload jpg images and those images are displayed in gallery.

Lama2

[INST]Find the issue in this code below. Explain your reason

import torch

torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

def run\_som\_func(a, b):

c = c\*2

c=a+b

print(c)

return c ^ 2

I get an error saying variable referred before[/INST] The issue in the code is that the variable `c` is being assigned a new value in each iteration of the function, but it is also being returned at the end of the function. This means that the last value of `c` is being returned, which is not what the code intends.

Here's the reason for the error:

In the line `c = c\*2`, `c` is being assigned a new value, which overwrites the previous value of `c`. This means that the value of `c` at the end of the function is `c = 2`, which is not what the code intends.

In the line `return c ^ 2`, the function is returning the value of `c`, which is now `2`. However, the code is trying to return the result of `c ^ 2`, which is `4`, but `c` is now `2`, so the result of `c ^ 2` is `0`, which is not what the code intends.

To fix the issue, the code should not assign a new value to `c` in the function, but instead return the original value of `c` at the end of the function. Here's the corrected code:

import torch

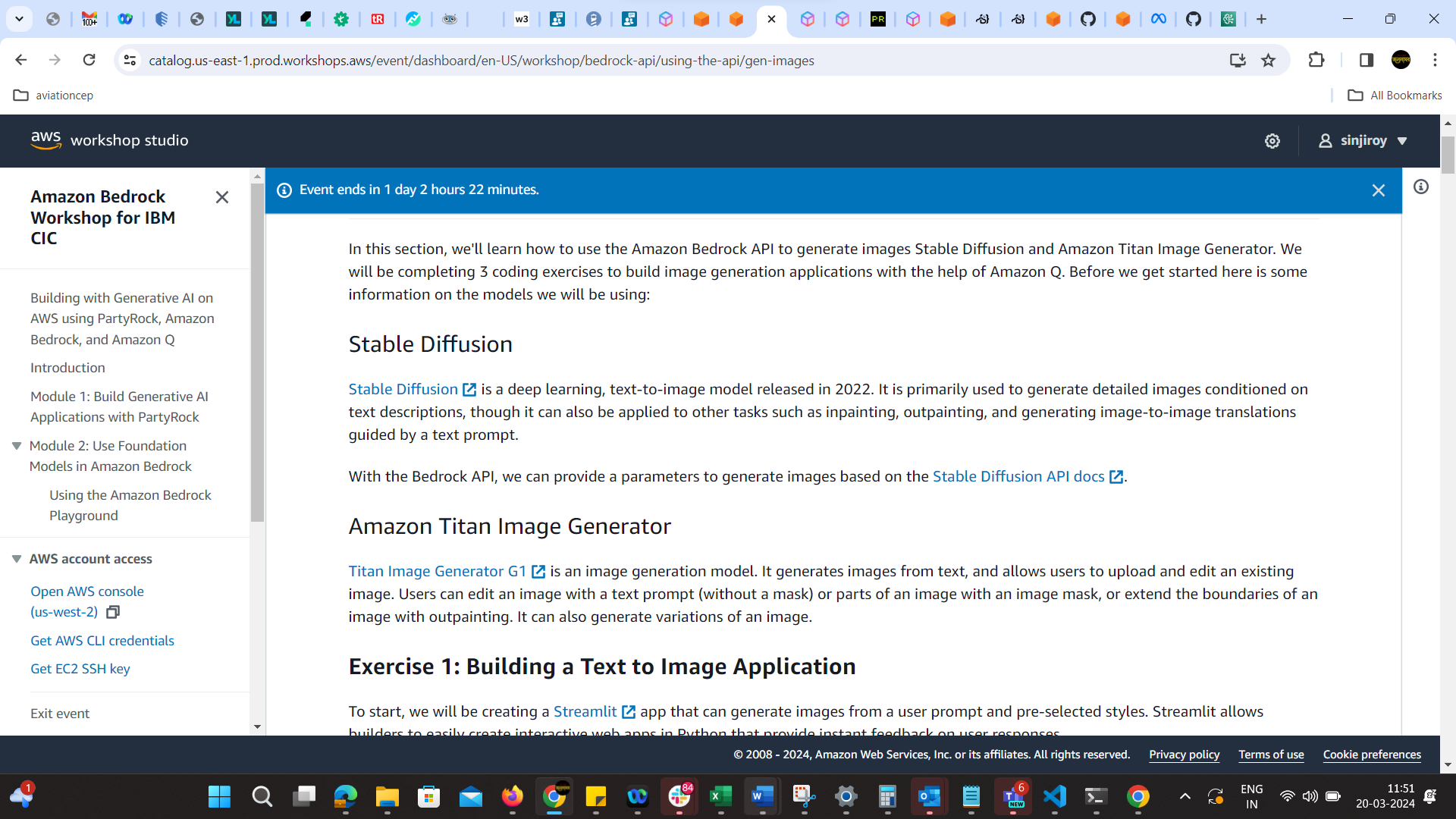
torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

def run\_som\_func(a, b):

c = a + b

print(c)

return c



import streamlit as st

import boto3

import json

import base64

import io

from PIL import Image

st.title("Building with Bedrock")  # Title of the application

st.subheader("Image Generation Demo")

REGION = "us-west-2"

# List of Stable Diffusion Preset Styles

sd\_presets = [

    "None",

    "3d-model",

    "analog-film",

    "anime",

    "cinematic",

    "comic-book",

    "digital-art",

    "enhance",

    "fantasy-art",

    "isometric",

    "line-art",

    "low-poly",

    "modeling-compound",

    "neon-punk",

    "origami",

    "photographic",

    "pixel-art",

    "tile-texture",

]

# Define bedrock

bedrock\_runtime = boto3.client(

    service\_name="bedrock-runtime",

    region\_name=REGION,

)

# Bedrock api call to stable diffusion

def generate\_image\_sd(text, style):

    """

    Purpose:

        Uses Bedrock API to generate an Image

    Args/Requests:

         text: Prompt

         style: style for image

    Return:

        image: base64 string of image

    """

    body = {

        "text\_prompts": [{"text": text}],

        "cfg\_scale": 10,

        "seed": 0,

        "steps": 50,

        "style\_preset": style,

    }

    if style == "None":

        del body["style\_preset"]

    body = json.dumps(body)

    modelId = "stability.stable-diffusion-xl"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("artifacts")[0].get("base64")

    return results

def generate\_image\_titan(text):

    """

    Purpose:

        Uses Bedrock API to generate an Image using Titan

    Args/Requests:

         text: Prompt

    Return:

        image: base64 string of image

    """

    body = {

        "textToImageParams": {"text": text},

        "taskType": "TEXT\_IMAGE",

        "imageGenerationConfig": {

            "cfgScale": 10,

            "seed": 0,

            "quality": "standard",

            "width": 512,

            "height": 512,

            "numberOfImages": 1,

        },

    }

    body = json.dumps(body)

    modelId = "amazon.titan-image-generator-v1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("images")[0]

    return results

def convert\_base64\_to\_image(base64\_string):

    img\_data = base64.b64decode(base64\_string)

    img = Image.open(io.BytesIO(img\_data))

    return img

model = st.selectbox("Select model", ["Stable Diffusion", "Amazon Titan"])

prompt = st.text\_input("Enter prompt")

style = st.selectbox("Select style", sd\_presets) if model=="Stable Diffusion" else None

def display\_image(encoded\_image):

   encoded\_image = encoded\_image.encode("utf-8")

   st.image(encoded\_image)

if st.button("Generate"):

    if model=="Stable Diffusion":

        image = generate\_image\_sd(prompt, style)

        imgm = convert\_base64\_to\_image(image)

        st.image(imgm)

    else:

        image = generate\_image\_titan(prompt)

        imgm = convert\_base64\_to\_image(image)

        st.image(imgm)

Inpainting full

import boto3

import json

import base64

import io

import os

from PIL import Image

import streamlit as st

from PIL import ImageOps

REGION = "us-west-2"

# Define bedrock

bedrock\_runtime = boto3.client(

    service\_name="bedrock-runtime",

    region\_name=REGION,

)

def inpaint\_mask(img, box):

    """Generates a segmentation mask for inpainting"""

    img\_size = img.size

    assert len(box) == 4  # (left, top, right, bottom)

    assert box[0] < box[2]

    assert box[1] < box[3]

    return ImageOps.expand(

        Image.new(mode="RGB", size=(box[2] - box[0], box[3] - box[1]), color="black"),

        border=(box[0], box[1], img\_size[0] - box[2], img\_size[1] - box[3]),

        fill="white",

    )

def gen\_mask\_from\_image(user\_image):

    img\_size = user\_image.size

    box = (

        (img\_size[0] - 300) // 2,

        img\_size[1] - 300,

        (img\_size[0] + 300) // 2,

        img\_size[1] - 200,

    )

    # Mask

    mask = inpaint\_mask(user\_image, box)

    return mask

def image\_to\_base64(img) -> str:

    """Convert a PIL Image or local image file path to a base64 string for Amazon Bedrock"""

    if isinstance(img, str):

        if os.path.isfile(img):

            print(f"Reading image from file: {img}")

            with open(img, "rb") as f:

                return base64.b64encode(f.read()).decode("utf-8")

        else:

            raise FileNotFoundError(f"File {img} does not exist")

    elif isinstance(img, Image.Image):

        print("Converting PIL Image to base64 string")

        buffer = io.BytesIO()

        img.save(buffer, format="PNG")

        return base64.b64encode(buffer.getvalue()).decode("utf-8")

    else:

        raise ValueError(f"Expected str (filename) or PIL Image. Got {type(img)}")

# Turn base64 string to image with PIL

def base64\_to\_pil(base64\_string):

    """

    Purpose:

        Turn base64 string to image with PIL

    Args/Requests:

         base64\_string: base64 string of image

    Return:

        image: PIL image

    """

    imgdata = base64.b64decode(base64\_string)

    image = Image.open(io.BytesIO(imgdata))

    return image

# Bedrock api call to stable diffusion

def sd\_inpaint\_image(change\_prompt, init\_image\_b64, mask):

    """

    Purpose:

        Uses Bedrock API to generate an Image

    Args/Requests:

         text: Prompt

         style: style for image

    Return:

        image: base64 string of image

    """

    body = {

        "text\_prompts": ([{"text": change\_prompt, "weight": 1.0}]),

        "cfg\_scale": 10,

        "init\_image": init\_image\_b64,

        "mask\_source": "MASK\_IMAGE\_BLACK",

        "mask\_image": image\_to\_base64(mask),

        "seed": 0,

        "start\_schedule": 0.6,

        "steps": 50,

    }

    body = json.dumps(body)

    modelId = "stability.stable-diffusion-xl"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("artifacts")[0].get("base64")

    return results

def titan\_inpaint\_image(change\_prompt, init\_image\_b64, mask):

    """

    Purpose:

        Uses Bedrock API to generate an Image using Titan

    Args/Requests:

         text: Prompt

    Return:

        image: base64 string of image

    """

    body = {

        "inPaintingParams": {

            "text": change\_prompt,

            "image": init\_image\_b64,

            "maskImage": image\_to\_base64(mask),

        },

        "taskType": "INPAINTING",

        "imageGenerationConfig": {

            "cfgScale": 10,

            "seed": 0,

            "quality": "standard",

            "width": 512,

            "height": 512,

            "numberOfImages": 1,

        },

    }

    body = json.dumps(body)

    modelId = "amazon.titan-image-generator-v1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("images")[0]

    return results

def inpaint\_image\_pipeline(user\_image, change\_prompt, mask, model):

    # Turn image to base64 string

    init\_image\_b64 = image\_to\_base64(user\_image)

    # Invoke Bedrock to inpaint the image

    if model == "Stable Diffusion":

        updated\_image = sd\_inpaint\_image(change\_prompt, init\_image\_b64, mask)

    elif model == "Amazon Titan":

        updated\_image = titan\_inpaint\_image(change\_prompt, init\_image\_b64, mask)

    # convert updated\_image to PIL image

    updated\_image = base64\_to\_pil(updated\_image)

    # save updated image

    updated\_image.save("inpainted\_image.png")

    return updated\_image

st.title("Building with Bedrock")  # Title of the application

st.subheader("Image Generation Demo - Inpainting")

model = st.selectbox("Select model", ["Stable Diffusion", "Amazon Titan"])

# Streamlit file uploader for only for images

user\_image = st.file\_uploader("Upload an image", type=["png", "jpg", "jpeg"])

# Get user prompt to change image

change\_prompt = st.text\_input("Enter prompt to change image")

# Create three columns, one to show the uploaded image, 2 for the mask, and 3 for the new image

col1, col2, col3 = st.columns(3)

# Show the uploaded image

if user\_image is not None:

    user\_image = Image.open(user\_image)

    col1.image(user\_image)

    if model == "Stable Diffusion":

        mask = Image.open("sd\_mask.png")

    else:

        mask = gen\_mask\_from\_image(user\_image)

    col2.image(mask)

    # Button to generate new image

    if st.button("Inpaint Image"):

        new\_image = inpaint\_image\_pipeline(user\_image, change\_prompt, mask, model)

        col3.image(new\_image)

else:

    col3.write("No image uploaded")

import boto3

import json

import base64

import io

import os

from PIL import Image

import streamlit as st

REGION = "us-west-2"

# Define bedrock

bedrock\_runtime = boto3.client(

    service\_name="bedrock-runtime",

    region\_name=REGION,

)

def image\_to\_base64(img) -> str:

    """Convert a PIL Image or local image file path to a base64 string for Amazon Bedrock"""

    if isinstance(img, str):

        if os.path.isfile(img):

            print(f"Reading image from file: {img}")

            with open(img, "rb") as f:

                return base64.b64encode(f.read()).decode("utf-8")

        else:

            raise FileNotFoundError(f"File {img} does not exist")

    elif isinstance(img, Image.Image):

        print("Converting PIL Image to base64 string")

        buffer = io.BytesIO()

        img.save(buffer, format="PNG")

        return base64.b64encode(buffer.getvalue()).decode("utf-8")

    else:

        raise ValueError(f"Expected str (filename) or PIL Image. Got {type(img)}")

# Turn base64 string to image with PIL

def base64\_to\_pil(base64\_string):

    """

    Purpose:

        Turn base64 string to image with PIL

    Args/Requests:

         base64\_string: base64 string of image

    Return:

        image: PIL image

    """

    imgdata = base64.b64decode(base64\_string)

    image = Image.open(io.BytesIO(imgdata))

    return image

# Bedrock api call to stable diffusion

def sd\_update\_image(change\_prompt, init\_image\_b64):

    """

    Purpose:

        Uses Bedrock API to generate an Image

    Args/Requests:

         text: Prompt

         style: style for image

    Return:

        image: base64 string of image

    """

    body = {

        "text\_prompts": ([{"text": change\_prompt, "weight": 1.0}]),

        "cfg\_scale": 10,

        "init\_image": init\_image\_b64,

        "seed": 0,

        "start\_schedule": 0.6,

        "steps": 50,

    }

    body = json.dumps(body)

    modelId = "stability.stable-diffusion-xl"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("artifacts")[0].get("base64")

    return results

def titan\_update\_image(change\_prompt, init\_image\_b64):

    """

    Purpose:

        Uses Bedrock API to generate an Image using Titan

    Args/Requests:

         text: Prompt

    Return:

        image: base64 string of image

    """

    body = {

        "imageVariationParams": {"text": change\_prompt, "images": [init\_image\_b64]},

        "taskType": "IMAGE\_VARIATION",

        "imageGenerationConfig": {

            "cfgScale": 10,

            "seed": 0,

            "quality": "standard",

            "width": 512,

            "height": 512,

            "numberOfImages": 1,

        },

    }

    body = json.dumps(body)

    modelId = "amazon.titan-image-generator-v1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("images")[0]

    return results

def update\_image\_pipeline(user\_image, change\_prompt, model):

    # Turn image to base64 string

    init\_image\_b64 = image\_to\_base64(user\_image)

    if model == "Stable Diffusion":

        updated\_image = sd\_update\_image(change\_prompt, init\_image\_b64)

    elif model == "Amazon Titan":

        updated\_image = titan\_update\_image(change\_prompt, init\_image\_b64)

    # convert updated\_image to PIL image

    updated\_image = base64\_to\_pil(updated\_image)

    # save updated image

    updated\_image.save("updated\_image.png")

    return updated\_image

st.title("Building with Bedrock")  # Title of the application

st.subheader("Image Generation Demo - Image to Image")

model = st.selectbox("Select model", ["Stable Diffusion", "Amazon Titan"])

# TODO insert your comments

# upload a image file

user\_image = st.file\_uploader("Upload an image")

# Get user prompt to change image

change\_prompt = st.text\_input("Enter prompt to change image")

col1, col2 = st.columns(2)  # Column 1 for input image, Column 2 for output image

# show user image

if user\_image is not None:

    user\_image = Image.open(user\_image)

    col1.image(user\_image)

    # Button to generate new image

    if col1.button("Update Image"):

        new\_image = update\_image\_pipeline(user\_image, change\_prompt, model)

        col2.image(new\_image)

else:

    col2.write("No image uploaded")

inpaint my code

import boto3

import json

import base64

import io

import os

from PIL import Image

import streamlit as st

from PIL import ImageOps

REGION = "us-west-2"

# Define bedrock

bedrock\_runtime = boto3.client(

    service\_name="bedrock-runtime",

    region\_name=REGION,

)

def inpaint\_mask(img, box):

    """Generates a segmentation mask for inpainting"""

    img\_size = img.size

    assert len(box) == 4  # (left, top, right, bottom)

    assert box[0] < box[2]

    assert box[1] < box[3]

    return ImageOps.expand(

        Image.new(mode="RGB", size=(box[2] - box[0], box[3] - box[1]), color="black"),

        border=(box[0], box[1], img\_size[0] - box[2], img\_size[1] - box[3]),

        fill="white",

    )

def gen\_mask\_from\_image(user\_image):

    img\_size = user\_image.size

    box = (

        (img\_size[0] - 300) // 2,

        img\_size[1] - 300,

        (img\_size[0] + 300) // 2,

        img\_size[1] - 200,

    )

    # Mask

    mask = inpaint\_mask(user\_image, box)

    return mask

def image\_to\_base64(img) -> str:

    """Convert a PIL Image or local image file path to a base64 string for Amazon Bedrock"""

    if isinstance(img, str):

        if os.path.isfile(img):

            print(f"Reading image from file: {img}")

            with open(img, "rb") as f:

                return base64.b64encode(f.read()).decode("utf-8")

        else:

            raise FileNotFoundError(f"File {img} does not exist")

    elif isinstance(img, Image.Image):

        print("Converting PIL Image to base64 string")

        buffer = io.BytesIO()

        img.save(buffer, format="PNG")

        return base64.b64encode(buffer.getvalue()).decode("utf-8")

    else:

        raise ValueError(f"Expected str (filename) or PIL Image. Got {type(img)}")

# Turn base64 string to image with PIL

def base64\_to\_pil(base64\_string):

    """

    Purpose:

        Turn base64 string to image with PIL

    Args/Requests:

         base64\_string: base64 string of image

    Return:

        image: PIL image

    """

    imgdata = base64.b64decode(base64\_string)

    image = Image.open(io.BytesIO(imgdata))

    return image

# Bedrock api call to stable diffusion

def sd\_inpaint\_image(change\_prompt, init\_image\_b64, mask):

    """

    Purpose:

        Uses Bedrock API to generate an Image

    Args/Requests:

         text: Prompt

         style: style for image

    Return:

        image: base64 string of image

    """

    body = {

        "text\_prompts": ([{"text": change\_prompt, "weight": 1.0}]),

        "cfg\_scale": 10,

        "init\_image": init\_image\_b64,

        "mask\_source": "MASK\_IMAGE\_BLACK",

        "mask\_image": image\_to\_base64(mask),

        "seed": 0,

        "start\_schedule": 0.6,

        "steps": 50,

    }

    body = json.dumps(body)

    modelId = "stability.stable-diffusion-xl"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("artifacts")[0].get("base64")

    return results

def titan\_inpaint\_image(change\_prompt, init\_image\_b64, mask):

    """

    Purpose:

        Uses Bedrock API to generate an Image using Titan

    Args/Requests:

         text: Prompt

    Return:

        image: base64 string of image

    """

    body = {

        "inPaintingParams": {

            "text": change\_prompt,

            "image": init\_image\_b64,

            "maskImage": image\_to\_base64(mask),

        },

        "taskType": "INPAINTING",

        "imageGenerationConfig": {

            "cfgScale": 10,

            "seed": 0,

            "quality": "standard",

            "width": 512,

            "height": 512,

            "numberOfImages": 1,

        },

    }

    body = json.dumps(body)

    modelId = "amazon.titan-image-generator-v1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("images")[0]

    return results

def inpaint\_image\_pipeline(user\_image, change\_prompt, mask, model):

    # Turn image to base64 string

    init\_image\_b64 = image\_to\_base64(user\_image)

    # Invoke Bedrock to inpaint the image

    if model == "Stable Diffusion":

        updated\_image = sd\_inpaint\_image(change\_prompt, init\_image\_b64, mask)

    elif model == "Amazon Titan":

        updated\_image = titan\_inpaint\_image(change\_prompt, init\_image\_b64, mask)

    # convert updated\_image to PIL image

    updated\_image = base64\_to\_pil(updated\_image)

    # save updated image

    updated\_image.save("inpainted\_image.png")

    return updated\_image

st.title("Building with Bedrock")  # Title of the application

st.subheader("Image Generation Demo - Inpainting")

model = st.selectbox("Select model", ["Stable Diffusion", "Amazon Titan"])

# Streamlit file uploader for only for images

user\_image = st.file\_uploader("Upload an image", type=["png", "jpg", "jpeg"])

# Get user prompt to change image

change\_prompt = st.text\_input("Enter prompt to change image")

# Create three columns, one to show the uploaded image, 2 for the mask, and 3 for the new image

col1, col2, col3 = st.columns(3)

# Show the uploaded image

if user\_image is not None:

    user\_image = Image.open(user\_image)

    col1.image(user\_image)

    if model == "Stable Diffusion":

        mask = Image.open("sd\_mask.png")

    else:

        mask = gen\_mask\_from\_image(user\_image)

    # TODO Finish App with Q

# Create three columns

col1, col2, col3 = st.columns(3)

# Show uploaded image

if user\_image is not None:

  col1.image(user\_image)

  # Generate mask

  mask = gen\_mask\_from\_image(user\_image)

  # Display mask

  col2.image(mask)

# Add button

if st.button("Inpaint"):

  # Run pipeline

  updated\_image = inpaint\_image\_pipeline(

    user\_image,

    change\_prompt,

    mask,

    model

  )

  # Display updated image

  col3.image(updated\_image)

select code and send to amazon Q and enter prompt

or alt+c to get codewhisperer help after typing your comment #enter a text input field

import boto3

import json

import time

# Setup bedrock

bedrock\_runtime = boto3.client(

    service\_name="bedrock-runtime",

    region\_name="us-west-2",

)

# Call Mistral model

def call\_mistral\_8x7b(prompt):

    prompt\_config = {

        "prompt": prompt,

        "max\_tokens": 4096,

        "temperature": 0.7,

        "top\_p": 0.8

    }

    body = json.dumps(prompt\_config)

    modelId = "mistral.mixtral-8x7b-instruct-v0:1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("outputs")[0].get("text")

    return results

# Call Mistral model

def call\_mistral\_7b(prompt):

    prompt\_config = {

        "prompt": prompt,

        "max\_tokens": 4096,

        "temperature": 0.7,

        "top\_p": 0.8

    }

    body = json.dumps(prompt\_config)

    modelId = "mistral.mistral-7b-instruct-v0:2"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("outputs")[0].get("text")

    return results

# Call AI21 labs model

def call\_ai21(prompt):

    prompt\_config = {

        "prompt": prompt,

        "maxTokens": 5147,

        "temperature": 0.7,

        "stopSequences": [],

    }

    body = json.dumps(prompt\_config)

    modelId = "ai21.j2-ultra-v1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("completions")[0].get("data").get("text")

    return results

def claude\_prompt\_format(prompt: str) -> str:

    # Add headers to start and end of prompt

    return "\n\nHuman: " + prompt + "\n\nAssistant:"

# Call Claude model

def call\_claude\_sonet(prompt):

    prompt\_config = {

        "anthropic\_version": "bedrock-2023-05-31",

        "max\_tokens": 4096,

        "messages": {

            "role": "user",

            "content": [

                {"type": "text", "text": prompt},

            ],

        },

    }

    body = json.dumps(prompt\_config)

    modelId = "anthropic.claude-3-sonnet-20240229-v1:0"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("completion")

    return results

# Call Claude model

def call\_claude(prompt):

    prompt\_config = {

        "prompt": claude\_prompt\_format(prompt),

        "max\_tokens\_to\_sample": 4096,

        "temperature": 0.5,

        "top\_k": 250,

        "top\_p": 0.5,

        "stop\_sequences": [],

    }

    body = json.dumps(prompt\_config)

    modelId = "anthropic.claude-v2:1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("completion")

    return results

# Call Cohere model

def call\_cohere(prompt):

    prompt\_config = {

        "prompt": prompt,

        "max\_tokens": 2048,

        "temperature": 0.7,

    }

    body = json.dumps(prompt\_config)

    modelId = "cohere.command-text-v14"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("generations")[0].get("text")

    return results

# Call Titan model

def call\_titan(prompt):

    prompt\_config = {

        "inputText": prompt,

        "textGenerationConfig": {

            "maxTokenCount": 4096,

            "stopSequences": [],

            "temperature": 0,

            "topP": 1,

        },

    }

    body = json.dumps(prompt\_config)

    modelId = "amazon.titan-text-express-v1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("results")[0].get("outputText")

    return results

def call\_llama2(prompt):

    prompt\_config = {

        "prompt": prompt,

        "max\_gen\_len": 2048,

        "top\_p": 0.9,

        "temperature": 0.2,

    }

    body = json.dumps(prompt\_config)

    modelId = "meta.llama2-13b-chat-v1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body["generation"].strip()

    return results

def summarize\_text(text):

    """

    Function to summarize text using a generative AI model.

    """

    prompt = f"Summarize the following text in 50 words or less: {text}"

    result = call\_titan(prompt)

    return result

def sentiment\_analysis(text):

    """

    Function to return a JSON object of sentiment from a given text.

    """

    prompt = f"Giving the following text, return a JSON object of sentiment analysis. text: {text} "

    result = call\_mistral\_8x7b(prompt)

    return result

def perform\_qa(question, text):

    """

    Function to perform a Q&A operation based on the provided text.

    """

    prompt = f"Given the following text, answer the question. If the answer is not in the text, 'say you do not know': {question} text: {text} "

    result = call\_claude(prompt)

    return result

if \_\_name\_\_ == "\_\_main\_\_":

    # Sample text for summarization

    text = "Amazon Bedrock is a fully managed service that offers a choice of high-performing foundation models (FMs) from leading AI companies like AI21 Labs, Anthropic, Cohere, Meta, Stability AI, and Amazon via a single API, along with a broad set of capabilities you need to build generative AI applications with security, privacy, and responsible AI. Using Amazon Bedrock, you can easily experiment with and evaluate top FMs for your use case, privately customize them with your data using techniques such as fine-tuning and Retrieval Augmented Generation (RAG), and build agents that execute tasks using your enterprise systems and data sources. Since Amazon Bedrock is serverless, you don't have to manage any infrastructure, and you can securely integrate and deploy generative AI capabilities into your applications using the AWS services you are already familiar with"

    print("\n=== Summarization Example ===")

    summary = summarize\_text(text)

    print(f"Summary:\n {summary}")

    time.sleep(2)

    print("\n=== Sentiment Analysis Example ===")

    sentiment\_analysis\_json = sentiment\_analysis(text)

    print(f"Sentiment\_Analysis JSON:\n{sentiment\_analysis\_json}")

    time.sleep(2)

    print("\n=== Q&A Example ===")

    q1 = "How many companies have models in Amazon Bedrock?"

    print(q1)

    answer = perform\_qa(q1, text)

    print(f"Answer: {answer}")

    time.sleep(2)

    q2 = "Can Amazon Bedrock support RAG?"

    print(q2)

    answer = perform\_qa(q2, text)

    print(f"Answer: {answer}")

    time.sleep(2)

    q3 = "When was Amazon Bedrock announced?"

    print(q3)

    answer = perform\_qa(q3, text)

    print(f"Answer: {answer}")

Retrieval Augmented Generation with Amazon Bedrock

Before diving into the RAG workflow, it's crucial to understand **embeddings.** An embedding is a way of representing documents as vectors in a high-dimensional space. These vectors capture the essence of the document's content in a form that machines can process. By converting text into embeddings, we enable the computer to 'understand' and compare different pieces of text based on their contextual similarities.

import json

import boto3

from langchain\_community.embeddings import BedrockEmbeddings

from langchain\_community.vectorstores import FAISS

REGION = "us-west-2"

# Setup bedrock

bedrock\_runtime = boto3.client(

    service\_name="bedrock-runtime",

    region\_name=REGION,

)

sentences = [

    # Pets

    "Your dog is so cute.",

    "How cute your dog is!",

    "You have such a cute dog!",

    # Cities in the US

    "New York City is the place where I work.",

    "I work in New York City.",

    # Color

    "What color do you like the most?",

    "What is your favourite color?",

]

def claude\_prompt\_format(prompt: str) -> str:

    # Add headers to start and end of prompt

    return "\n\nHuman: " + prompt + "\n\nAssistant:"

# Call Claude model

def call\_claude(prompt):

    prompt\_config = {

        "prompt": claude\_prompt\_format(prompt),

        "max\_tokens\_to\_sample": 4096,

        "temperature": 0.5,

        "top\_k": 250,

        "top\_p": 0.5,

        "stop\_sequences": [],

    }

    body = json.dumps(prompt\_config)

    modelId = "anthropic.claude-v2:1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("completion")

    return results

def rag\_with\_bedrock(query):

    embeddings = BedrockEmbeddings(

        client=bedrock\_runtime,

        model\_id="amazon.titan-embed-text-v1",

    )

    local\_vector\_store = FAISS.from\_texts(sentences, embeddings)

    docs = local\_vector\_store.similarity\_search(query)

    context = ""

    for doc in docs:

        context += doc.page\_content

    prompt = f"""Use the following pieces of context to answer the question at the end.

    {context}

    Question: {query}

    Answer:"""

    return call\_claude(prompt)

query = "What city I am currently working in?"

print(query)

print(rag\_with\_bedrock(query))

chat with pdf

import json

import os

import boto3

from langchain\_community.embeddings import BedrockEmbeddings

from langchain\_community.vectorstores import FAISS

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain\_community.document\_loaders import (

    UnstructuredFileLoader,

)

import os

REGION = "us-west-2"

# Setup bedrock

bedrock\_runtime = boto3.client(

    service\_name="bedrock-runtime",

    region\_name="us-west-2",

)

def chunk\_doc\_to\_text(doc\_loc: str):

    loader = UnstructuredFileLoader(doc\_loc)

    docs = loader.load()

    text\_splitter = RecursiveCharacterTextSplitter(

        chunk\_size=1000, chunk\_overlap=20

    )

    texts = text\_splitter.split\_documents(docs)

    return texts

def claude\_prompt\_format(prompt: str) -> str:

    # Add headers to start and end of prompt

    return "\n\nHuman: " + prompt + "\n\nAssistant:"

# Call Claude model

def call\_claude(prompt):

    prompt\_config = {

        "prompt": claude\_prompt\_format(prompt),

        "max\_tokens\_to\_sample": 4096,

        "temperature": 0.5,

        "top\_k": 250,

        "top\_p": 0.5,

        "stop\_sequences": [],

    }

    body = json.dumps(prompt\_config)

    modelId = "anthropic.claude-v2:1"

    accept = "application/json"

    contentType = "application/json"

    response = bedrock\_runtime.invoke\_model(

        body=body, modelId=modelId, accept=accept, contentType=contentType

    )

    response\_body = json.loads(response.get("body").read())

    results = response\_body.get("completion")

    return results

def rag\_with\_bedrock(query):

    embeddings = BedrockEmbeddings(

        client=bedrock\_runtime,

        model\_id="amazon.titan-embed-text-v1",

    )

    pdf\_loc = "well\_arch.pdf"

    if os.path.exists("local\_index"):

        local\_vector\_store = FAISS.load\_local("local\_index", embeddings, allow\_dangerous\_deserialization=True)

    else:

        texts = chunk\_doc\_to\_text(pdf\_loc)

        local\_vector\_store = FAISS.from\_documents(texts, embeddings)

        local\_vector\_store.save\_local("local\_index")

    docs = local\_vector\_store.similarity\_search(query)

    context = ""

    for doc in docs:

        context += doc.page\_content

    prompt = f"""Use the following pieces of context to answer the question at the end.

    {context}

    Question: {query}

    Answer:"""

    return call\_claude(prompt)

query = "What can you tell me about AWS Well-Architected Framework?"

print(query)

print(rag\_with\_bedrock(query))